# Design Patterns

Gerwin van Dijken (gerwin.vandijken@inholland.nl)

# Program term 1.4

# Strategy pattern

- Makes it possible to dynamically change the strategy (algorithm/behaviour) of an object

- This 'changeable' behaviour is <u>decoupled</u> from the object

The **<u>Strategy Pattern (GoF)</u>**: *'defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.'*

# Strategy pattern - 3 parts

- <u>Strategy</u>: the interface that defines how the algorithm needs to be called

- <u>Concrete Strategy</u>: the implementation of the strategy

- <u>Context</u>: the object that contains the Concrete Strategy (via interface reference)
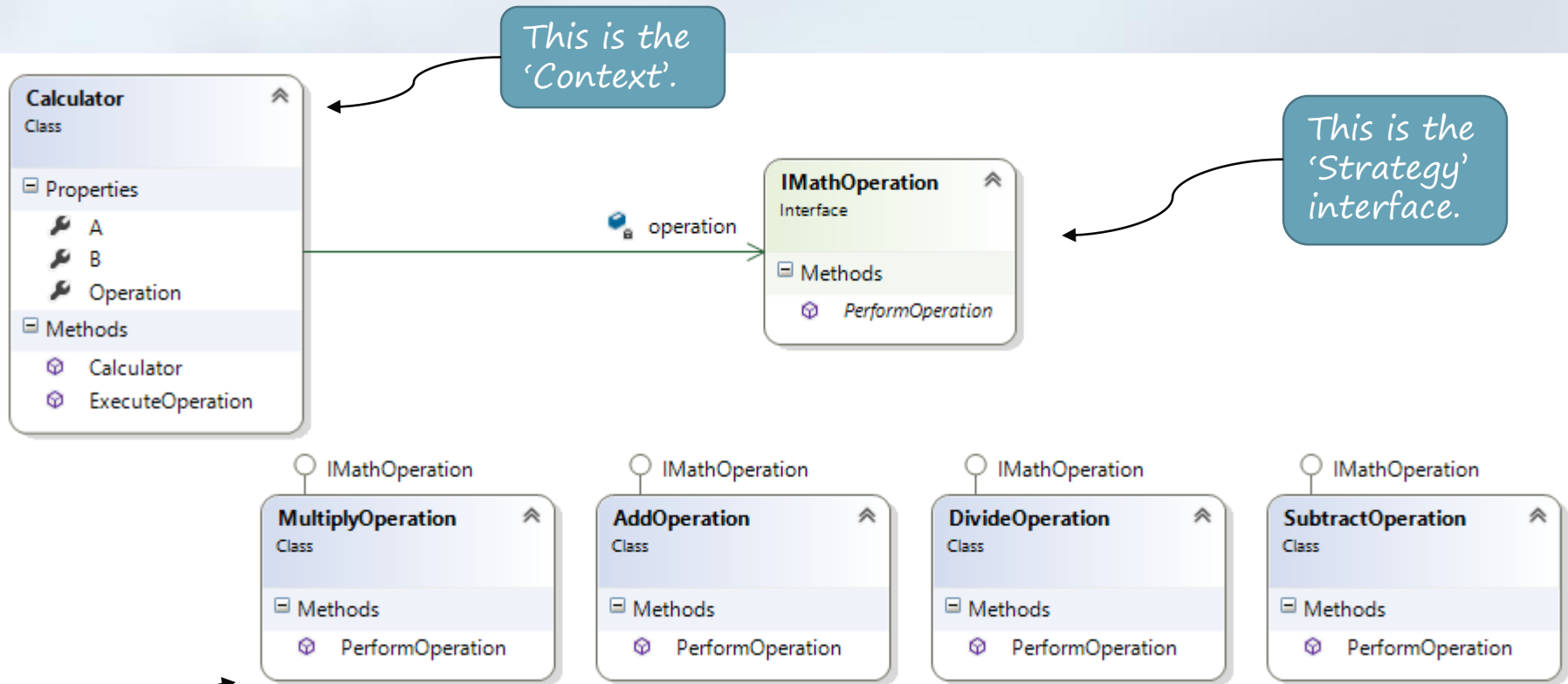
# Strategy pattern



'Context' contains a reference to a Strategy (interface) object.
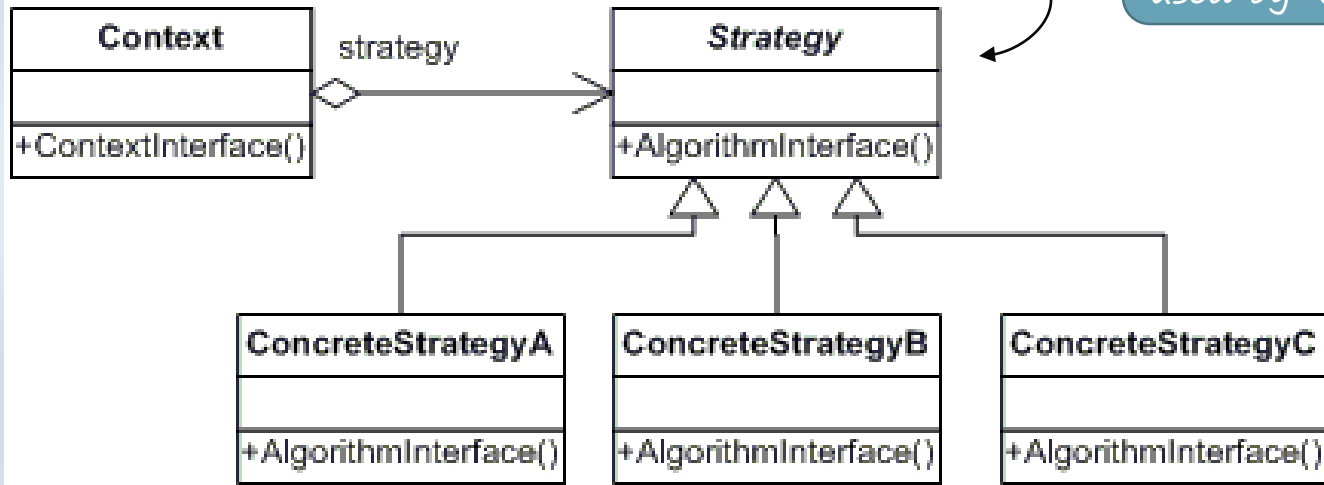
'Strategy' defines an interface that is used by 'Context'.

The ConcreteStrategy classes implement the interface (algorithm).

# Strategy pattern – an example



This is the 'Context'.

This is the 'Strategy' interface.

These are 4 concrete 'Strategy' classes.

**Calculator**
Class

Properties
- A
- B
- Operation

Methods
- Calculator
- ExecuteOperation

operation

**IMathOperation**
Interface

Methods
- *PerformOperation*

IMathOperation

**MultiplyOperation**
Class

Methods
- PerformOperation

IMathOperation

**AddOperation**
Class

Methods
- PerformOperation

IMathOperation

**DivideOperation**
Class

Methods
- PerformOperation

IMathOperation

**SubtractOperation**
Class

Methods
- PerformOperation

# Strategy pattern – a

```csharp
public interface IMathOperation
{
  double PerformOperation(
        double a, double b);
}
```

The 'Strategy' interface.

The 4 concrete 'Strategy' classes.

```csharp
public class AddOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a + b;
  }
}


public class SubtractOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a - b;
  }
}


public class MultiplyOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a * b;
  }
}


public class DivideOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a / b;
  }
}
```

# Strategy pattern – an example

```csharp
public class Calculator
{
    private IMathOperation operation;

    public double A { get; set; }
    public double B { get; set; }

    public IMathOperation Operation
    {
        get { return operation; }
        set { operation = value; }
    }

    // constructor
    public Calculator()
    {
        A = B = 0;

        // default behaviour
        operation = new AddOperation();
    }

    public double ExecuteOperation()
    {
        return operation.PerformOperation(A, B);
    }
}
```

Reference to 'Strategy' interface.

With this property 'Operation' the behaviour can be changed.

Processing is done by the current concrete (operation) object.

# Strategy pattern – an

file:///C:/Users/gerwin/Document...

```
[default]
Input: 35 and 2, result: 37
[multiply]
Input: 35 and 2, result: 70
[divide]
Input: 35 and 2, result: 17.5
```
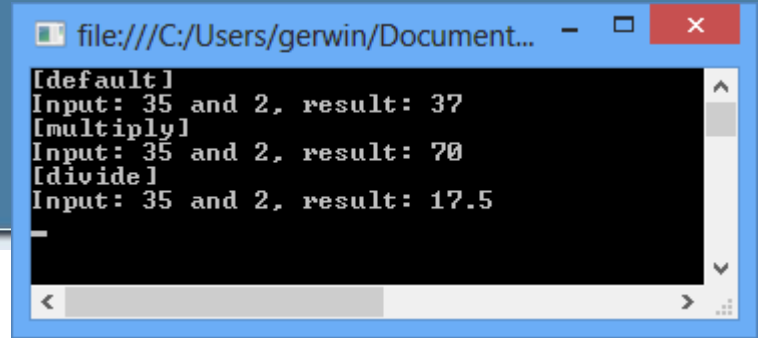
```csharp
static void Main(string[] args)
{
  Calculator calculator = new Calculator();
  calculator.A = 35;
  calculator.B = 2;

  Console.WriteLine("[default]");
  double result = calculator.ExecuteOperation();
  Console.WriteLine(String.Format("Input: {0} and {1}, result: {2}",
          calculator.A, calculator.B, result));

  Console.WriteLine("[multiply]");
  calculator.Operation = new MultiplyOperation();
  result = calculator.ExecuteOperation();
  Console.WriteLine(String.Format("Input: {0} and {1}, result: {2}",
          calculator.A, calculator.B, result));

  Console.WriteLine("[divide]");
  calculator.Operation = new DivideOperation();
  result = calculator.ExecuteOperation();
  Console.WriteLine(String.Format("Input: {0} and {1}, result: {2}",
          calculator.A, calculator.B, result));

  Console.ReadKey();
}
```

*Here the 'strategy' is called.*

*By changing the operation ('on the fly'), the behaviour of the calculator will change.*

# Advantages of Strategy pattern

- 1. You can change behaviour 'at runtime'
  *(this can not be done with derived classes... like 'Conway Game of Life' classes)*

- 2. You can 'store' (and change) multiple kinds of behaviours in one class
  *(multiple inheritance is not possible in C#...)*
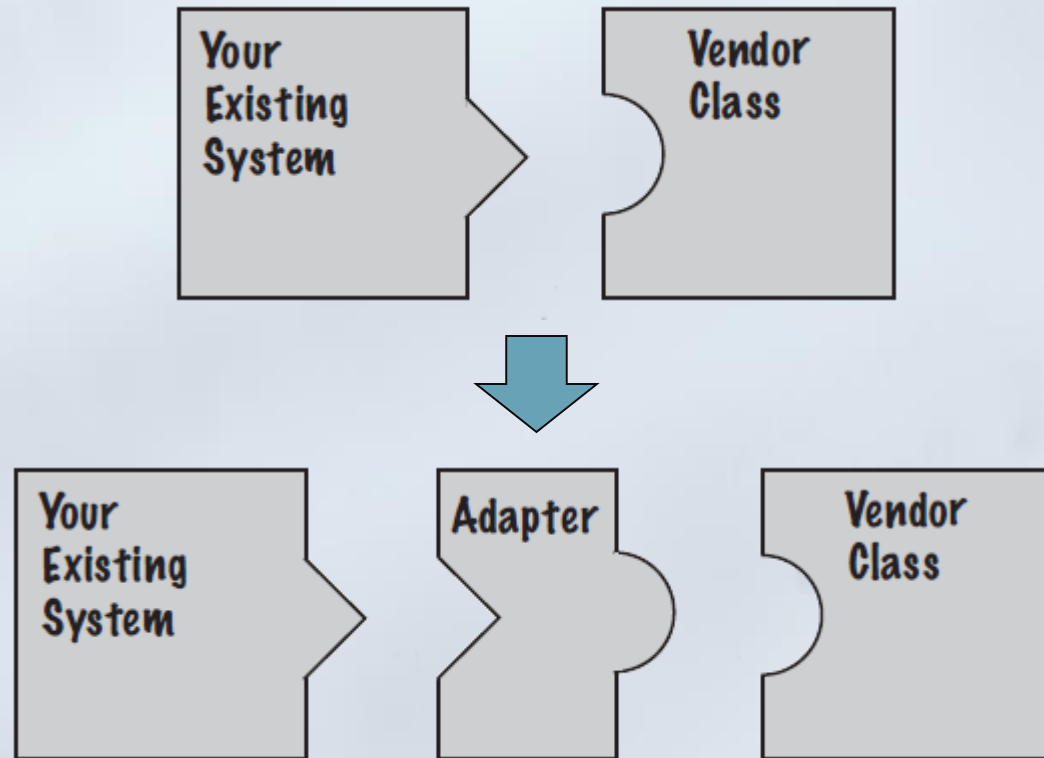
# Adapter pattern

■ Convert an interface into another interface, in order to make 2 existing systems compatible (allow them to work together)



The <u>Adapter Pattern (GoF)</u>: 'converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.'

# Adapter pattern

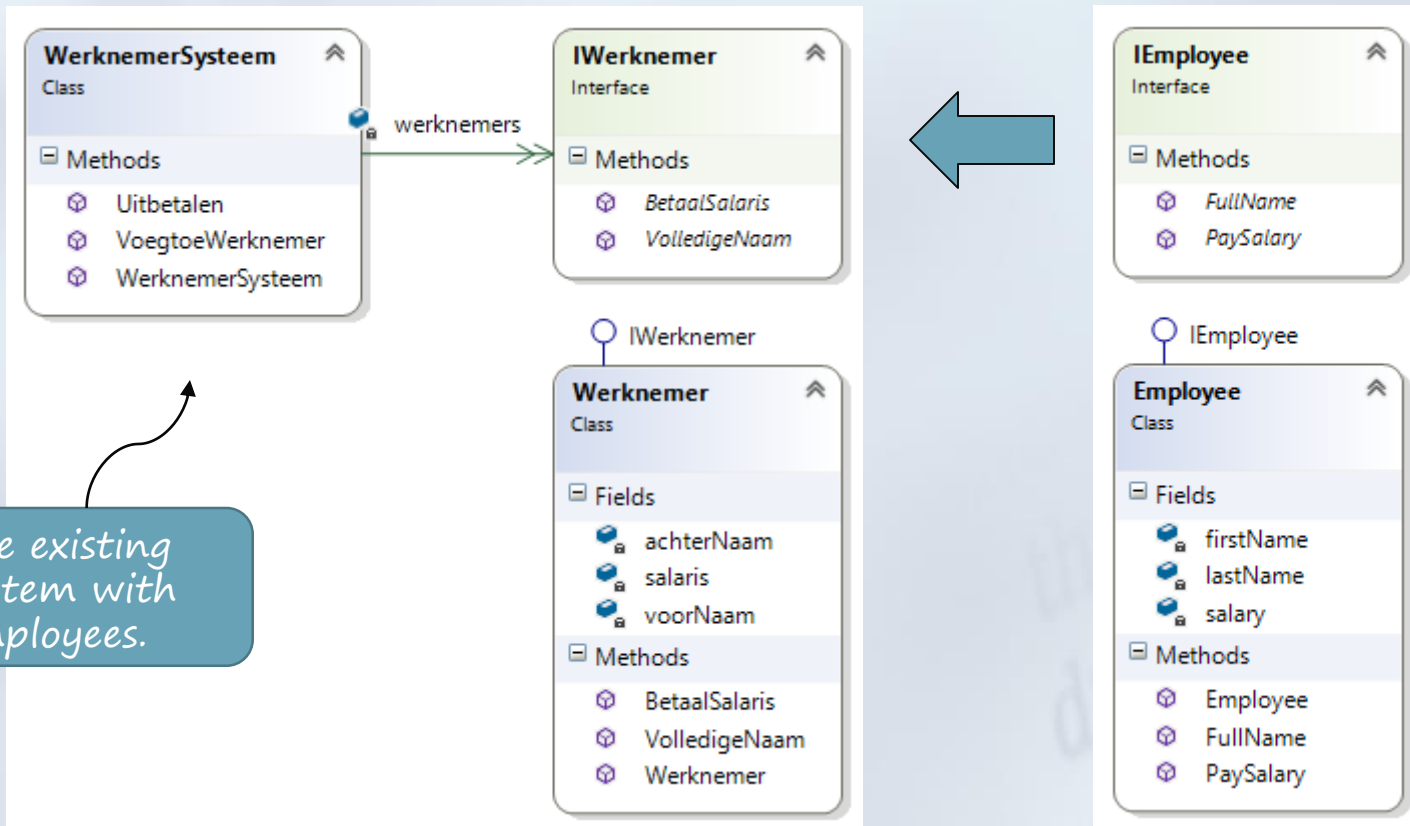- Allow existing software to work with other software
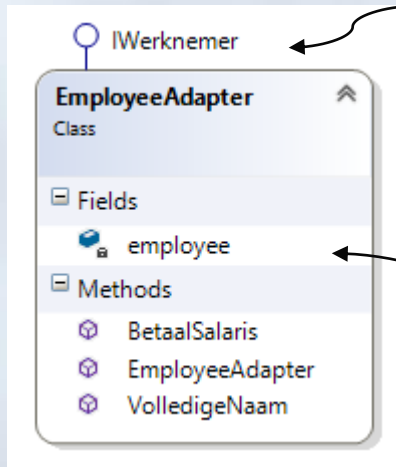
# Adapter pattern – an example

- An employee system with ... <u>Dutch</u> employees (IWerknemer)

- Another existing system must be added, containing... <u>English</u> employees (IEmployee)

- We don't want to change the existing systems

# Adapter pattern – an example



We want to add IEmployee to the systeem.

The existing system with employees.

# Adapter pattern – an example

The Adapter behaves like an IWerknemer...

...but the actions are executed by an IEmployee.

IEmployee is passed to the constructor of the Adapter.

IWerknemer

**EmployeeAdapter**
Class

Fields
- employee

Methods
- BetaalSalaris
- EmployeeAdapter
- VolledigeNaam

```
class EmployeeAdapter      IWerknemer
{
  private IEmployee employee;

  public EmployeeAdapter(IEmployee employee)
  {
    this.employee = employee;
  }

  public string VolledigeNaam()
  {
    return employee.FullName();
  }

  public void BetaalSalaris()
  {
    employee.PaySalary();
  }
}
```

# Adapter pattern - main

```csharp
static void Main(string[] args)
{
  IWerknemer werknemer1 = new Werknemer("Kees", "van Kralingen", 2500);
  IWerknemer werknemer2 = new Werknemer("Karel", "van Dijk", 2800);
  IWerknemer werknemer3 = new Werknemer("Pieter", "de Boer", 2200);

  WerknemerSysteem systeem = new WerknemerSysteem();
  systeem.VoegtoeWerknemer(werknemer1);
  systeem.VoegtoeWerknemer(werknemer2);
  systeem.VoegtoeWerknemer(werknemer3);

  // add employee using an Adapter
  Employee employee1 = new Employee("Sam", "Potter", 3000);
  //systeem.VoegtoeWerknemer(employee1);  // this does not compile...
  systeem.VoegtoeWerknemer(new EmployeeAdapter(employee1));

  systeem.Uitbetalen();

  Console.ReadKey();
}
```

*Here we use an adapter, in order to allow the system to work with (English) IEmployee (employee is disguised as IWerknemer).*

```csharp
class WerknemerSysteem
{
  List<IWerknemer> werknemers = new List<IWerknemer>();

  // ...

  public void Uitbetalen()
  {
    foreach (IWerknemer werknemer in werknemers)
      werknemer.BetaalSalaris();
  }
}
```

# Summary

- Strategy Pattern: delegate behaviour in order to make it interchangeable

- Adapter Pattern: convert a 'new' interface to an existing interface in order to let both systems work together

# Assignments

- BlackBoard: 'Week 4 assignments'